C++ Services Library 6

Contents

This chapter is organized as follows:

C++ Services Library Framework

Together, the classes and data types defined in the C++ Services Library provide an application framework - the framework in which your Rational Rose RealTime application runs.

In general, the framework defines the skeleton of a real-time application: **messaging**, **timing**, **dynamic structure**, **concurrency**, **event based processing**, **platform independence**. You provide the classes, capsules, and protocols which are specific to your system.

The Big Advantage

Rational Rose RealTime lets you develop using state diagrams and structure diagrams which are automatically converted to C++ and placed in a framework that provides critical real-time system services.

Before you start developing, the key to using the services provided by the framework, is to understand how your application will integrate into the C++ Services Library skeleton. The framework provides:

- Communication services are the basic mechanism for using message-based communication via ports.
- Timing service provides general purpose timing facilities.
- Frame service is used to gain control over the dynamic structure in a model.
- Log service is a general purpose logging service.
- Exception service provides the ability to define custom policies to recover from exceptions.

Services are explained by introducing the general concepts related to the service followed by the classes that are used to implement that service. You should become familiar with the C++ syntax and notational conventions used in these sections as well as the *Services Library Class Reference* on page 233.

Message Processing

Events and Messages

An event is a message arriving on a capsule's port. Message-based communication is the basic mechanism for communication between capsules. Both synchronous and asynchronous communication are supported allowing a variety of different interaction semantics to be represented. Messages are also used by the Services Library to communicate with the capsules in the model.

The pre-defined capsule instance variable msg contains a pointer to the current message just received by the behavior. It is defined at the highest scope in the behavior. A message has three attributes:

- A signal that succinctly conveys the application-specific “meaning” of the message.
- A priority that indicates the “urgency” of the message. The priority of a message is determined by the sender.
- An optional data attribute, which contains additional information. This attribute can consist of an arbitrarily complex composite data object.

Processing Overview

The Services Library does not preempt capsule processing. The heart of the Services Library is a controller object that dispatches messages to capsules. Its basic mode of operation is to take the next message from the outstanding message queue and deliver it to the destination capsule for processing. When it delivers the message, it invokes the destination capsule's state machine to process the message.

Control is not returned to the Services Library until the capsule's transition has completed processing the message. Each capsule processes only one message at a time. It processes the current message to the completion of the transition chain (for example, guard, exit, transition, Choice Point, exit, and entry) and then returns control to the Services Library and waits for the next message. This is referred to as run-to-completion semantics. Typically, transition code segments are short, and result in rapid handling of messages.

Single and Multi-Threaded Message Processing

The Services Library runs in a loop executed by a system controller object. This loop waits for messages and delivers them, one at a time, to capsules for processing. Each physical thread in a Rose RealTime model has its own controller object and its own set of message queues. Messages that cross threads are placed in a special queue and picked up by the receiving thread in its processing.

The model is first initialized by queueing a special system-level message (the initialization message) for the top-level capsule. This causes initialization messages to be queued for all fixed capsules contained inside the top-level capsule. This continues recursively for all contained fixed capsules, so that all the fixed capsules in the model (those that aren't contained in optional capsules) are initialized.

After the initialization message is queued, the controller object enters its main processing loop (the **mainLoop** function). In **mainLoop**, it takes the next highest priority message from the message queues and delivers it to the receiver capsule and invokes that capsule's behavior to process the message. During start-up, the highest priority message on the queue of the main thread will be the initialization message. When a capsule processes the initialization message, the capsule's initial transition segment is executed.

When the capsule has completed processing a message, it returns control to the controller. The controller continues this loop until there are no more messages to be processed. At that point, it waits for a message from a timer or another physical thread in the model.

Introduction to Threads

A capsule can be thought of as having its own logical thread of control, and operating independently of other capsules, as if each capsule had its own dedicated processor. These independent capsules synchronize to perform higher-level scenarios through message-passing. One capsule sends a message to another capsule allowing the other capsule to update its state based on this outside stimulus. In practice, most Rose RealTime models run on a machine with a single processor, or possibly in a distributed environment, with a few processors. Capsules must share the single processor in some manner.

Types of Concurrency

The underlying operating system provides preemption to allow concurrent programs to share the processor so that each program is guaranteed to get some processing time depending on the prioritization of the programs, and any program that blocks does not stop processing of other programs. Many operating systems support one or both of the following forms of concurrency:

1. A heavy-weight unit of concurrency (usually referred to as a process), which has its own memory space, is completely separate from other processes (for integrity), and which communicates with other processes through special mechanisms (shared memory, sockets, signals, and so forth). Processes usually have a significant amount of protection such that if one process crashes it does not affect any other processes.
2. A light-weight unit of concurrency, referred to as a thread (or task on most RTOSs), shares a common memory space with other threads, and is not as robust (can be corrupted by other threads). Threads do not have as much protection as processes. Depending on the type of failure, an error in one thread may affect other threads.

Mapping Capsules to Threads

Rational Rose RealTime allows designers to make use of the underlying multi-tasking operating system so that the processing of a capsule on one thread does not block the processing of capsules on other threads. Designers can specify the physical operating system threads onto which the capsules will be mapped at run-time. In a system with only one thread, there are situations where a single capsule transition can block other capsules from running, such as if the capsule invokes a blocking system call. By

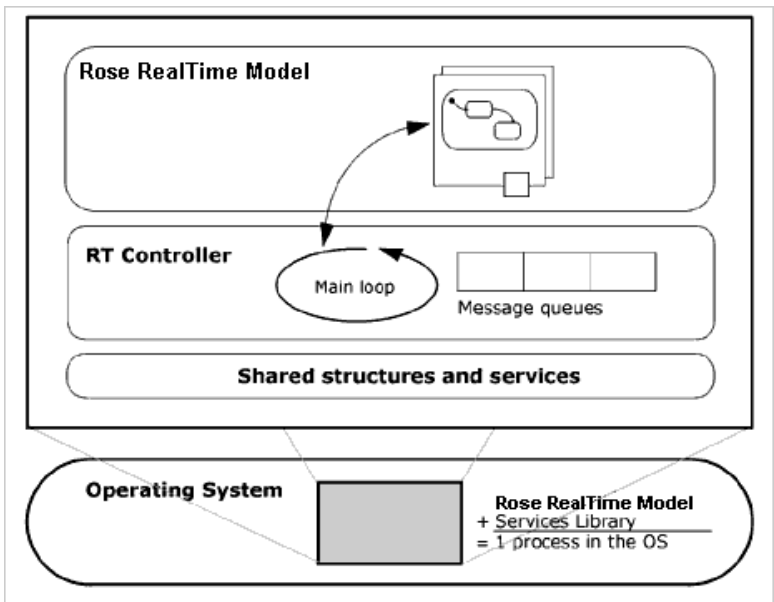
placing some capsules in different threads, the designer can avoid the problems that arise from these situations, and make better use of the underlying processor. Not every capsule should run on a separate thread. For most capsules, it is sufficient to leave them in one thread and allow the Services Library controller to invoke their behavior as messages arrive.

Capsules with transitions that may block, or that have excessively long processing times, should be placed on separate threads. Deciding which capsules need to execute in different threads is a matter for design consideration.

Single-Threaded Services Library

The use of threads is not supported for certain targets, and may not be desirable for some applications. There is a single-threaded version of the Services Library, which is used for these situations. In the single-threaded model there is a single controller object that is responsible for queueing and delivering messages among capsules. The main processing loop runs inside this object. The single-threaded Services Library has the basic structure shown in Figure 12.

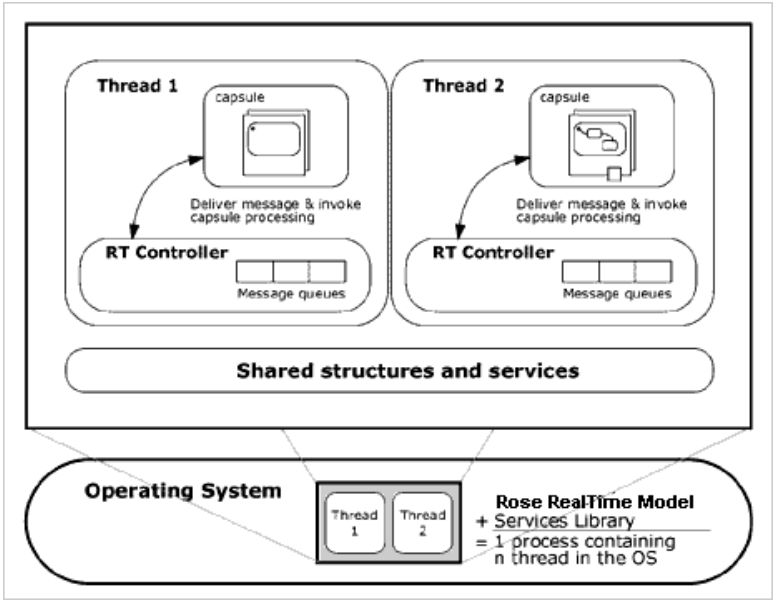
Figure 12 Single-Threaded Services Library



Multi-Threaded Services Library

Capsules can belong to different logical threads. Logical threads are mapped to a set of concurrent physical threads defined by the developer. No other capsules in a thread can execute until the currently executing capsule returns control to the main loop of that thread (except for the case of invoke). However, other capsules on other physical threads may be executing simultaneously (at least, from the designer's perspective). The operating system is responsible for switching control among active physical threads. The operating system may preempt one physical thread in the middle of execution to switch to another physical thread. Each thread can be assigned a separate priority, so that the designer has some control over the scheduling. In the multi-threaded model there is a separate controller object for each physical thread. This controller object contains the basic message delivery and processing loop. The basic structure of the multi-threaded Services Library is shown in Figure 13.

Figure 13 Multi-Threaded Services Library



Naming Considerations

The C++ Services Library contains class names and operation names that may not exactly line up with the terminology used in the rest of the product. This is a holdover from previous versions of the Services Library, which was based on terminology used in the Real-time Object-Oriented Modeling (ROOM) method. Briefly, the changes in terminology are:

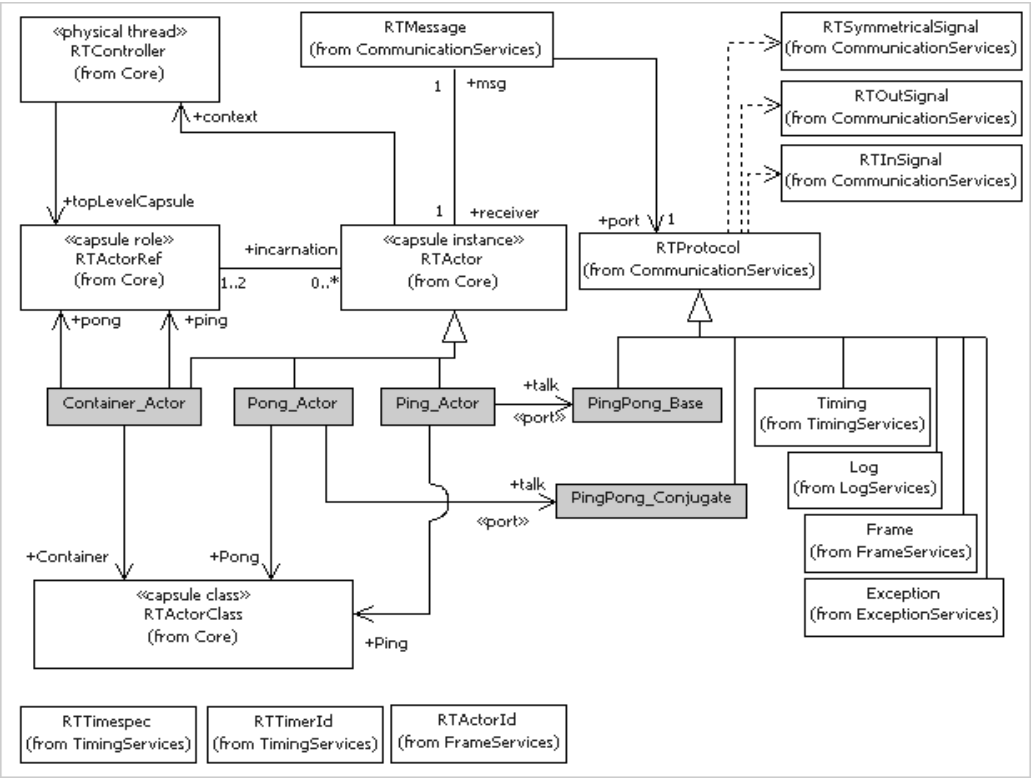
- actor = capsule instance
- actor reference = capsule role
- actor class = capsule
- SAP = unwired port for accessing a service
- SPP = unwired port for providing a service

C++ Services Library Framework

The capsules, capsule roles, protocols, ports and classes in a Rational Rose RealTime model will be generated to C++ code and integrated into the C++ Services Library framework. The class diagram below shows how a set of generated model elements integrate within the framework.

The white boxes are predefined classes in the C++ Services Library and the grey boxes are classes generated from a *Framework Sample Model* on page 105.

Figure 14 C++ Services Library Framework



From this simplified class diagram, observe the following:

- The high level view of the C++ Services Library classes and their relationships.
- How your application level modeling elements integrate into this framework (grey boxes).
- The relationships between your modeling elements and the framework.

Because most modeling elements will become subclasses of framework base classes, elements will have access to operations and attributes that are defined in the base classes. Here are the main relationships that you should understand:

- Capsules become subclasses of RTActor
- Special Overrideable capsule class operations
- Capsule class information is stored in instances of RTActorClass
- Capsule roles are attributes of type RTActorRef
- Protocols become two classes - Base and Conjugate
- Ports are Protocol type attributes in RTActor subclasses

- Signals become operations in protocol classes
- Capsule roles are place holders for zero or more capsule instances
- Capsule instances have access to a RTMessage object
- Capsule instances have access to their controller

Capsules Become Subclasses of RTActor

All capsules are generated as subclasses of RTActor. This common base class contains state machine processing and messaging behavior that is used by all capsules.

Capsule state machines, operations, and attributes are included in the generated RTActor subclass. Since there can possibly be many instances of the same capsule in an application, capsule class information is kept separate from the RTActor instances, in a RTActorClass metaclass.

You can access public operations of RTActor within a capsule's behavior. For example it is common to have the following C++ code in a capsule state transition where the operations RTActor::getError and RTActor::context can be used because they are defined on the RTActor class:

```
switch( getError() )
{
   case RTController::noConnect:
   log.log("Unable to send message");
      break;

   default:
      log.show( "Unexpected error sending to peer: ");
      log.show( context()->strerror() );
      log.cr();
      break;
}
```

Special Overrideable Capsule Class Operations

There are two special operations that are defined as virtual functions on the C++ Services Library root class, RTActor, of all capsules. These functions can be used to customize the capsule's response to certain conditions.

- RTActor::unexpectedMessage
- RTActor::logMsg

Note: See the RTActor class reference for details on how to use these operations.

Capsule Class Information is Stored in Instances of RTActorClass

Characteristics common to all capsules, for example a name and external interfaces, are kept in a RTActorClass structure. All operations in the C++ Services Library which require you to specify a capsule class will take a parameter of type RTActorClass. There is only one instance of a RTActorClass for each capsule, whereas there are usually many RTActor capsule instances. The obvious advantage is that all capsule instances of the same capsule can share the capsule information stored in the RTActorClass.

The RTActorClass structures are named exactly as the capsules in your model. So if you have a capsule called **Device** in your model you can directly refer to this class in your model as **Device**. For example a common usage of RTActorClass is in the Frame::incarnate method where you can specify which type of capsule to incarnate into an optional capsule role. You specify the type by using the name of the capsule directly in the operation call. The first parameter specifies the capsule role and the second the capsule class:

```
frame.incarnate( device, Printer );
```

Capsule Roles are Attributes of Type RTActorRef

A capsule’s structure is defined by a number of communicating capsule roles. In the framework the capsule roles become attributes of type RTActorRef in the containing capsule.

The attribute name is kept the same as the role name, which means that you can reference a capsule role by name in detail C++ code of the containing capsule.

Protocols Become Two Classes: Base and Conjugate

For each protocol two classes are generated to represent the base and conjugate protocol roles. The protocol role classes are generated as subclasses of the root protocol class and are filled in with operations specific to the in and out signals defined in the protocol roles.

In the C++ Services Library Framework diagram you can see that the **Ping_Actor** and **Pong_Actor** each have a port of the same type but assigned to different protocol roles.

Ports are Protocol Type Attributes in RTActor Subclasses

A port on a capsule becomes an attribute in the generated RTActor subclass. The port attribute has the same name as the port in the model. The port attribute will be a subclass of the common base class, **RTProtocol**. It be directly referenced by name in a capsule's C++ detail code. Here a port named talk is defined on a capsule.

The port definition in the capsule class:

```
public:
   // {{{RME protocolClass 'Commands' port 'talk'
   Commands::Base talk;
```

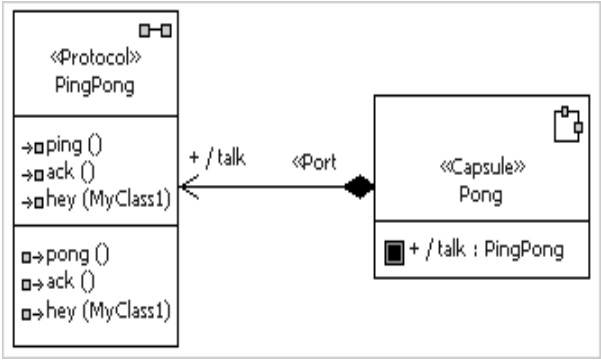
The port being referenced in detail code on the capsule:

```
talk.ping().send();
```

Signals Become Operations in Protocol Classes

Each signal defined in a protocol is generated as an operation with the same name as the signal in the generated protocol class. The return value of the operation dictates the actions that can be performed with the signal. Incoming signals **RTInSignal** and outgoing signals **RTOutSignal** will obviously differ in allowed actions.

Figure 15 PingPong Protocol and Talk Port



From this class diagram you see that the **Pong** capsule has a port named talk of type **PingPong**. The port is not conjugated. The unconjugated generated protocol class PingPong::Base will have operations for each signal and will allow you to reference them from the talk attribute generate on the **Pong_Actor** class.

```
talk.pong().send();
```

This line of code calls the generated signal operation on the port, which returns a RTOutSignal object. Then the common action on an out signal is to send it.

Capsule Roles are Place Holders for Zero or More Capsule Instances

Capsule roles, or RTActorRef classes, are basically place holders for capsule instances. Replicated capsule roles are place holders for multiple instances of compatible RTActor subclasses.

See *RTActorRef* on page 240 for more information on the main uses of capsule roles in your model.

Multiple Containment

Often a capsule instance will only run in the context of a single capsule role, but with multiple containment, a single instance can exist in two or more capsule roles simultaneously.

Capsule Instances Have Access to a RTMessage Object

An RTActor class has access to the current message, RTMessage, that it received. You will often want to access this message in your capsule C++ detail code.

Capsule Instances Have Access to their Controller

Each capsule instance has access to the controller for the thread on which it is running. The RTController class provides several operations that can be useful in a capsule's implementation.

Framework Sample Model

This is the model which was used as an example of how elements from a model will integrate into the C++ Services Library Framework. The grey boxes in the diagram on the C++ Services Library Framework page show classes generated from this model:

Figure 16 Ping Pong Model Class Diagram

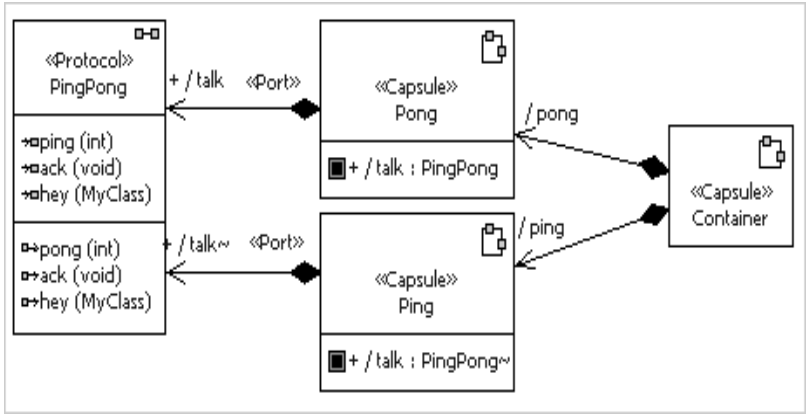
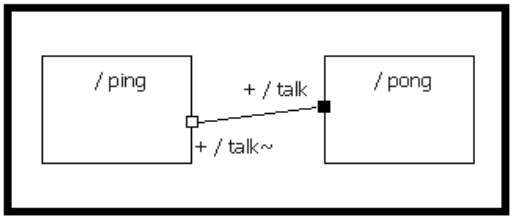


Figure 17 Container Capsule Structure Diagram



Log Service

Implementation classes

Log

Concepts

The System Log is a stream of ASCII text in which system or application events can be recorded. Currently all log output is directed to **stderr**.

Execution speed is affected since each write to the log involves an output system call, which is a relatively expensive operation.

Communication Services

Implementation classes

RTProtocol, RTOutSignal, RTInSignal, RTSymmetricSignal

Concepts

This fundamental service provides most of the standard communication models prevalent in concurrent software system design including asynchronous messaging and rendez-vous like synchronous inter-capsule communication.

The Communication Service is accessed by referencing, by name, a port (which will be a subclass of the RTProtocol class) with the appropriate operation. The port name is the user defined name of the port declared in the model. The named port is generated as a member of the capsule containing the port.